arXiv:1203.0259v1 [cs.DS] 1 Mar 2012

A REARRANGEMENT STEP WITH POTENTIAL USES IN PRIORITY QUEUES

BORIS ALEXEEV AND M. BRIAN JACOKES

Abstract. Link-based data structures, such as linked lists and binary search trees, have many well-known rearrangement steps allowing for efficient implementations of insertion, deletion, and other operations. We describe a rearrangement primitive designed for link-based, heap-ordered priority queues in the comparison model, such as those similar to Fibonacci heaps or binomial heaps.

In its most basic form, the primitive rearranges a collection of heap-ordered perfect binary trees. Doing so offers a data structure control on the number of trees involved in such a collection, in particular keeping this number logarithmic in the number of elements. The rearrangement step is free from an amortized complexity standpoint (using an appropriate potential function).

1. Introduction

The priority queue, or *heap*, is perhaps the simplest data structure beyond stacks and FIFO queues. Originally designed as part of the sorting algorithm *heapsort* [Wil64], heaps are now used in a variety of algorithms, particularly graph-theoretic ones (such as Dijkstra's single-source shortest-paths algorithm). In its most basic form, a heap represents a collection $\mathcal{P}$ of elements from a totally-ordered set, together with the operations insert and delete-min; the latter returns the name of the least element after deleting it from $\mathcal{P}$. More useful heap implementations will include the operations decrease-key, meld, delete, and make-queue, which we do not describe in detail here.

The importance and utility of heaps stems from fast access to the current minimum element, unlike the fixed access patterns of stacks and queues. Simultaneously, the absence of operations like iteration through $\mathcal{P}$ in sorted order allows heaps to answer queries faster than full-fledged binary search trees. Many heap implementations have been developed, including the original binary heap [Wil64], binomial heap [Vui78], and Fibonacci heaps [FT87]. The basic atom of these implementations is the concept of a *heap-ordered tree*, which is a rooted tree such that each element is no greater than any of its children. In such a tree, clearly the smallest element is the root.

Most implementations of heaps in the tradition of binomial heaps maintain a collection of heap-ordered trees represented explicitly, that is, with pointers. (An important exception is binary heaps which maintain a single heap-ordered tree represented implicitly in an array.) In order to perform updates and answer queries efficiently, there are a few primitive operations that rearrange elements in a manner preserving the heap-ordering. Two of the most important are *bubble-up* (also known as up-heap) and *bubble-down* (also known as down-heap), which address local inconsistencies in the heap-ordering at a leaf or root, respectively. There are also other primitives, such as the procedure for merging two binomial trees.

We introduce a new rearrangement primitive that rearranges a collection of heap-ordered trees. In short, given three perfect heap-ordered binary trees, the rearrangement creates one larger perfect heap-ordered tree and two smaller ones.

Date: February 24, 2013.
Key words and phrases. Priority queue, heap, rearrangement, perfect binary tree.

2. Rearrangement step

Recall that a binary tree is *perfect* (also called *full*) if all leaves are at the same depth. Such a tree will necessarily contain a number of nodes one less than a power of two.

A *rearrangement step* is a method of taking three perfect binary trees of the same size (say, of height h) and producing one perfect binary tree of height $h+1$ and two perfect binary trees of height $h-1$, in constant time. This can be accomplished by comparing the three roots, selecting the smallest one, removing it from its tree (leaving two subtrees), and making it the root of a new larger tree with the other two trees as children. See Figure 1 for visual accompaniment.

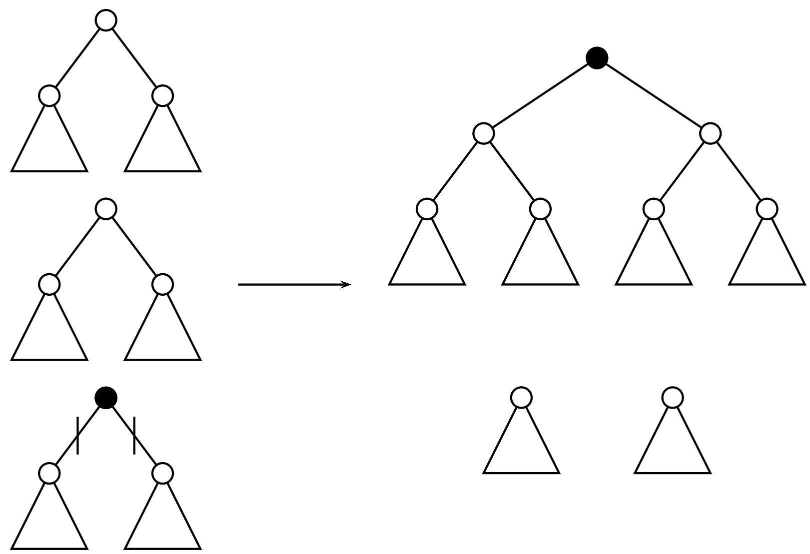

Figure 1. A rearrangement step: three trees of height h are rearranged to make one tree of height $h+1$ and two trees of height $h-1$ by selecting the smallest root (shaded) and rearranging children.

Suppose a heap is implemented by maintaining a collection of perfect, heap-ordered trees [SS85, BSG03]. Using the rearrangement step iteratively if necessary, the data structure may ensure that no more than two trees of any given height are present in the collection. Because the number of elements in each tree grows exponentially with the height, this ensures that no more than logarithmically-many trees are in the collection overall. Other operations, such as delete-min or meld, are thus easier to implement in logarithmic time.

From the point of view of amortized complexity [Tar85], the rearrangement step pays for itself. More specifically, consider a *potential function* φ that is the sum of the heights of the trees in a collection of perfect binary trees. Evidently, this potential function goes down by one every rearrangement step. Therefore, the amortized time complexity $t_{\text{amortized}} = t_{\text{actual}} + \varphi$ (normalized so that each rearrangement step takes one unit of t_{actual}) does not change when a rearrangement step is performed. Since in any sequence of operations beginning with an empty queue, the initial value of φ (namely, zero) is at most the final value of φ, the amortized time $t_{\text{amortized}}$ is an upper bound on the actual time t_{actual}. Of course, for this analysis to be useful, the other operations must play well with the potential function. In our case, straightforward implementations of the heap operations do not modify φ by much.

Of course, the rearrangement step need not be limited in its application to perfect binary trees only. However, its implementation in that case is most straightforward so we limit our description to that situation.

3. Applications

The rearrangement step described in this paper was also discovered by Claus Jensen, independently of these authors. Furthermore, Elmasry, Jensen, and Katajainen have used the step to implement a priority queue that guarantees $O(1)$ worst-case cost per insert and $O(\log n)$ worst-case cost per delete, where n is the number of elements stored. [EJK10, EJK12]

More specifically, those authors analyze a number system based on powers of two minus one (that is, the size of a perfect heap). A straightforward use of the rearrangement step would then allow each digit to be at most two; the *skew binary number system* has this property. However, in order to guarantee *worst-case* times rather than simply amortized times, the authors allow each digit to be at most four. Our writeup complements that of Elmasry, Jensen, and Katajainen by isolating the rearrangement primitive, which is used inline in their paper under the name *fix*.

Thus this rearrangement step, in addition to being a theoretical addition to the basic link mechanisms used in the existing plethora of link-based priority queues, has already found applications.

References

[BSG03] S. Bansal, S. Sreekanth, and P. Gupta, *M-heap: a modified heap data structure*, Internat. J. Found. Comput. Sci. **14** (2003), no. 3, 491–502.

[EJK10] Amr Elmasry, Claus Jensen, and Jyrki Katajainen, *The magic of a number system*, Proceedings of the 5th international conference on Fun with algorithms (Berlin, Heidelberg), FUN'10, Springer-Verlag, 2010, pp. 156–165.

[EJK12] ———, *Two skew-binary numeral systems and one application*, Theory Comput. Syst. **50** (2012), 185–211.

[FT87] Michael L. Fredman and Robert Endre Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach. **34** (1987), no. 3, 596–615.

[SS85] Thomas Strothotte and Jörg-Rüdiger Sack, *Heaps in heaps*, Proceedings of the sixteenth Southeastern international conference on combinatorics, graph theory and computing (Boca Raton, Fla., 1985), vol. 49, 1985, pp. 223–235.

[Tar85] Robert Endre Tarjan, *Amortized computational complexity*, SIAM J. Algebraic Discrete Methods **6** (1985), no. 2, 306–318.

[Vui78] Jean Vuillemin, *A data structure for manipulating priority queues*, Comm. ACM **21** (1978), no. 4, 309–315.

[Wil64] J. W. J. Williams, *Algorithm 232: Heapsort*, Comm. A.C.M. **7** (1964), no. 6, 347–348.